CS1Bh Practical 2

Inside the Java Virtual Machine

This is an individual practical exercise which requires you to submit some files electronically. A system which measures software similarity will be used to compare all of the submissions in order to detect unreasonable levels of collaboration. Your attention is drawn to the guidelines on plagiarism presented in Appendix A of the Computer Science 1 course guide.

This practical has been issued on Thursday 13th of February 2003. The deadline for your solutions is **Thursday 27th February 2003 at 5pm**. Your solutions must be submitted electronically from your DICE account with the `handin` command (which is a short-hand version of the `submit` command). No alternative methods of submission will be accepted. Specifically, submissions via email will not be accepted and submissions on floppy disk will not be accepted. No late submissions will be accepted.

To submit your files, execute the `handin` command *exactly* as it is shown in this document. In particular, make sure that you use the correct practical identifiers, filenames and extensions. Remember that all names are case-sensitive. Failure to execute the proper submission command cannot be investigated after the practical deadline. Execute the `handin` command from a shell window (*not* the KDE command “Run Command” window), so you can see any error messages which result. If in doubt, consult a laboratory demonstrator for assistance.

You can repeat the `handin` command to resubmit improved versions of your work; later submissions will override earlier ones. But do not resubmit after the deadline: your work would be treated as a late submission.

The four practical exercises in Computer Science 1Bh are equally weighted and together they constitute 25% of the final mark for the course. For each practical, a mark out of 25 is recorded.

**Getting started**

Before starting the practical you should make your own copy of the files in the directory `/home/cs1/CS1/BhPracs/PracB2`. You can do this efficiently by issuing the command:

```bash
   cp -r /group/teaching/cs1/Bh/Pracs/VirtualMachine ~
```
Your home directory will now contain a private directory called VirtualMachine with subdirectories PartA, PartB and PartC. These directories contain (identical copies of) seven Java source files and some Java bytecode test files.

GUI.java  Parser.java  SyntaxError.java
Instruction.java  RunTimeError.java  VirtualMachine.java
StackFrame.java  test0  test1
test2  test3  test4
test5  test6

Working on your computer at home

You can make a copy of the VirtualMachine directory on a floppy disk to take the files home. Insert a formatted DOS disk in the disk drive of the machine which you are using and issue the following command to copy all of the necessary files across.

```
mcopy -/ ˜/VirtualMachine a:
```

Alternatively, if you have Internet access from your machine at home then you can download the practical templates from this Web address

```
http://www.dcs.ed.ac.uk/teaching/cs1/CS1/Bh/Pracs/VirtualMachine/
```

Java byte code instructions

The subset of Java byte code used in this practical contains the following instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>goto ⟨ln⟩</td>
<td>Take line number ⟨ln⟩ as the next instruction</td>
</tr>
<tr>
<td>iconst ⟨c⟩</td>
<td>Push the constant ⟨c⟩ on top of the stack</td>
</tr>
<tr>
<td>iadd</td>
<td>Pop the top two integers from the stack, add them, and push the result back onto the top of the stack</td>
</tr>
<tr>
<td>if_icmp ⟨op⟩ ⟨ln⟩</td>
<td>If the top two operands are related by ⟨op⟩ go to line ⟨ln⟩</td>
</tr>
<tr>
<td>iinc ⟨i⟩ ⟨v⟩</td>
<td>Add value ⟨v⟩ to the current value of variable ⟨i⟩</td>
</tr>
<tr>
<td>iload ⟨i⟩</td>
<td>Push the value of variable number ⟨i⟩ on top of the stack</td>
</tr>
<tr>
<td>invokestatic ⟨s⟩</td>
<td>Invoke the static method called ⟨s⟩</td>
</tr>
<tr>
<td>ireturn</td>
<td>Exit the method leaving an integer result on top of the stack</td>
</tr>
<tr>
<td>istore ⟨i⟩</td>
<td>Pop the stack, using the result as the new value of variable ⟨i⟩</td>
</tr>
</tbody>
</table>

As an example, consider the operation if_icmplt 5. This will perform the following sequence of operations:

1. pop the stack to obtain the right operand of the comparison operation;
2. pop the stack again to obtain the left operand of the comparison operation;
3. evaluate the expression left < right to get a logical result
   - if the result was true then set the program counter to line 5;
   - if the result was false then proceed to the next instruction.
Part A — Constructing byte code instructions

You are provided with a class Instruction. You should first read the Instruction class file and ensure that you understand it. You are to design Java classes which can represent Java byte code instructions as subclasses of the Instruction class. To do this for this part you will need to do the following four things.

1. Decide on the fields which each class requires to store the operands of the relevant byte code instruction.
2. Complete the implementation of the constructor for the class. The header of the constructor has been provided for you but you will need to fill in the statements in the body of the constructor. (This will depend on the fields which you defined for the class in step 1 above.)
3. Implement the toString() method for the class (which will also depend on the fields of the class).
4. Set the visibility of the fields and the toString() method (are they public or are they private?)

When you complete this part you will have a way of representing Java Byte Code instructions as objects which can be stored in a vector. This vector makes up the program which you will go on to execute in Part B adding method invocation as an additional feature in Part C.

As an example, consider the input file containing the following instructions:

0 iconst_4
1 istore_1
2 ireturn

This byte code describes placing the value 4 on the stack (iconst_4) and then storing that value in the first variable (istore_1).

Each line of byte code is to be represented as an instruction. Here we have three lines of byte code so we will have three instructions. Each instruction has the following format ([line number, instruction code [arg1, arg2, ...]]) i.e. it contains a line number and a code followed by zero or more arguments. In the above example each instruction has one argument each—namely 4 for the first instruction and 1 for the second. Each of the two instructions is then stored in a vector representing the program.

The objects which are stored in the vector containing the program would be those shown below.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>new Instruction.IConst(0, 4)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>new Instruction.IStore(1, 1)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>new Instruction.IReturn(2)</td>
<td></td>
</tr>
</tbody>
</table>

There are three instructions with different numbers of parameters. The constructor represents the instruction code. Instruction.IConst is the constructor for "iconst_" and "istore_" is represented by Instruction.IStore. The first parameter refers to the line number, either 0, 1 or 2. The second parameter if present refers to the operand of the instruction. The operand of iconst_is 4 and the operand of istore_is 1.
Getting started on Part A

The Goto class is provided in the class Instruction as an example to help get you started. This has one integer field (jump). The constructor stores the line number and jump label which it is given (lineno is a field of the class Instruction). The toString() method uses both of these fields to make a printable representation of the instruction.

```java
public static class Goto extends Instruction {

    // Index of the instruction to jump to
    private int jump;

    // The constructor for the class
    public Goto(int lineno, int jump) {
        // Store the values which are passed in to this constructor
        this.linen = lineno;
        this.jump = jump;
    }

    public String toString() {
        // Return a string representation of this instruction
        return "" + lineno + ": goto " + jump;
    }

    public int evaluate(VirtualMachine machine) throws RunTimeError {
        // Return the new value of the program counter
        return jump;
    }
}
```

Ignore the evaluate() method for now. You do not need to implement the evaluate() methods until Part B.

For Part A of this practical exercise you are to complete the corresponding class definitions for the following classes which represent the Java Byte Code instructions which were described on page 2.

- IConst
- IAdd
- IInc
- ILoad
- IStore
- IReturn
- IICmp

Ignore the InvokeStatic class for now. You do not need to implement the InvokeStatic class until Part C.
Compiling and running your program

To compile your program issue the command

```bash
javac *.java
```

To run your program on the input test program `test0` issue the following command.

```bash
java GUI test0
```

Change the line to `java GUI test1` to run `test1` instead and similarly for other tests. When you are happy that your program is working as it should, submit it for assessment by issuing this command from the `PartA` subdirectory

```bash
handin B2a Instruction.java
```

Caution: issue the `handin` command exactly as shown above. If you issue the command incorrectly your work may not be submitted for assessment.

What you should see when you have finished Part A

When you have finished Part A of this practical you should be able to see the byte code program correctly printed in the Program area of the virtual machine simulator interface. The screenshot below illustrates this for the input file `test0`.
Part B — Byte code interpretation

In this part you are to extend your program from Part A with an `evaluate()` method for each instruction except `invokestatic`. You will be continuing to develop the file `Instruction.java` which you worked on in Part A. Make a copy of that file in your PartB directory before progressing further. Issue the following command.

```
cp ~/VirtualMachine/PartA/Instruction.java ~/VirtualMachine/PartB/
```

You may like to adjust the header to reflect that this file will now be your solution to Part B.

Evaluating instructions changes the state of the Java Virtual Machine. We can picture the state of the Java Virtual Machine as a stack of frames, each of which contains storage for the operand stack and local variables of a method and a program counter. Objects stored on the Virtual Machine call stack are of class `StackFrame`.

```
public class StackFrame {
    // Class constructor
    public StackFrame(Vector program,
                       String name);

    // The evaluation stack
    // which stores integers
    public Stack stack;

    // The current method
    public Vector program;

    // The maximum number of variables
    public static final int MEMSIZE = 8;

    // The memory of the machine
    public Integer[] memory;

    // Has the program finished?
    public boolean finished;

    // The program counter (pc)
    public int pc;
}
```

Evaluating an instruction may place a value on the top of the stack, or write a value to a location in memory, or update the program counter (which points to the next instruction to be executed).

This class uses objects which are of class `java.util.Stack`, `java.util.Vector` and an array of objects of class `java.lang.Integer`. You should consult past lecture notes and the course textbook to see examples of programs which use objects such as these. You may also find it helpful to read the Java documentation on-line to learn about the methods which you can invoke on objects of these classes.
The `VirtualMachine` class defines the state of the virtual machine. It is summarised below.

```java
public class VirtualMachine {
    // The run-time call stack
    // which stores StackFrames
    public Stack callStack;

    // Accessor methods for the top frame
    // Accessor methods for the program counter
    public int pc();
    public void setPc(int address);

    // Accessor methods for the finished flag
    public boolean finished();
    public void setFinished(boolean b);

    // Accessor methods for the memory
    public int memSize();
    public Integer memory(int address);
    public void setMemory(int address, Integer value);

    // Accessor methods for the operand stack
    // of the current method
    public Stack stack();

    // The instructions of the current method
    public Vector getProgramText();

    // Load a new method from a file
    public Vector load(String theFilename);

    // The method which gets the index
    // of the next instruction (pc + 1)
    public int getNextIndex();
}
```

Implement the `evaluate()` methods for the classes which you defined in Part A. Each `evaluate()` method has the following form.

```java
public int evaluate(VirtualMachine machine) throws RunTimeError
```

This method is passed the virtual machine as its parameter and it returns the new value of the program counter as its integer result. Upon discovering that the instruction cannot be executed for some reason, the method throws a Java exception of class `RunTimeError`. Recall that the syntax for specifying that an exception is thrown is

```java
throw new RunTimeError(" some diagnostic error message here ");
```

Refer to the lecture note “Compilation II: Compiling Expressions” for more examples of exceptions.
What you should see when you have finished Part B

When you have finished Part B of this practical you should be able to execute the byte code program either by single-stepping through it with the [Step] button or running it to completion with the [Run] button. As you single-step through the program you will see values being written into memory and you will see the stack growing and shrinking. The stack is printed ‘upside down’ so that it grows downwards and shrinks upwards (like an icicle). The top of the stack is therefore the value which is at the bottom of the list of numbers in the Stack area of the virtual machine simulator interface.

The screenshot below illustrates the result of executing the simulator on the input file test4.

![Java Virtual Machine Simulator](image-url)

Submitting your work for assessment

When you are happy that your program is working as it should, submit it for assessment by issuing this command from the PartB subdirectory:

```
handin B2b Instruction.java
```

!! Caution: issue the handin command exactly as shown above. If you issue the command incorrectly your work may not be submitted for assessment.
Part C — Method invocation

In this part you are to extend your program from Part B with an implementation of the `toString()` method and the `evaluate()` method for the `invokestatic` instruction. You will be continuing to develop the file `Instruction.java` which you worked on in Part B. Make a copy of that file in your PartC directory before progressing further. Issue the following command.

```bash
cp ~/VirtualMachine/PartB/Instruction.java ~/VirtualMachine/PartC/
```

You may like to adjust the header to reflect that this file will now be your solution to Part C.

We have ignored the `invokestatic` instruction until now. We now implement this instruction. The instruction `invokestatic test4` invokes the method from the file named `test4`. This is slightly different from the Java Byte Code program which we have seen in lectures because there the `invokestatic` instruction refers to a method by number and not by name.

As with all input/output operations, this operation could fail because the file does not exist or is not readable. In this case the `load()` method will return a `null` pointer. This is a run-time error in the virtual machine.

If the pointer which is returned by the invocation of the `load()` method is non-null then we need to construct a new stack frame and put it on top of the call stack. That method will be the next method to be executed.

We make the following simplifying assumption about the methods which are invoked:

```
| every method which is invoked has no parameters and returns an integer as its result |
```

Of course we can program methods in Java which accept numerous parameters and return results other than integers but our virtual machine accepts only a small subset of the Java byte code language and does not allow us to program these more complicated methods.

When we invoke a new method in the Java Virtual Machine we always begin to execute it at the beginning: we can never jump into the middle of a method. The first instruction in a method is always numbered zero.

When you implement the `invokestatic` instruction you may need to revise the implementation of the `ireturn` instruction which you implemented in Part B. Consider these two instructions as a pair and ask if you are returning the virtual machine to the correct state when you return from a method which you have invoked.
What you should see when you have finished Part C

When you have finished Part C of this practical you should be able to execute the byte code program either by single-stepping through it with the [Step] button or running it to completion with the [Run] button. As you load new methods you will see that the call stack records which methods are being invoked and that the program area is updated to show the code of the method which is currently being executed.

The screenshot below illustrates a mid-point during the execution of the input file test6. At this point the method test4 has just been invoked. It appears on the call stack and the code of the method is displayed in the Program window.

![Java Virtual Machine Simulator](image)

Submitting your work for assessment

When you are happy that your program is working as it should, submit it for assessment by issuing this command from the PartC subdirectory

```
handin B2c Instruction.java
```

!! *Caution: issue the handin command exactly as shown above. If you issue the command incorrectly your work may not be submitted for assessment.*
Undertaking this practical exercise

As with all programming exercises, it is possible that this exercise may take longer to complete than you might expect. Each exercise in Computer Science 1 introduces new concepts and encourages you to develop new programming skills. Acquiring these new skills will come more quickly for some people than for others. Therefore it is important that you begin work on this exercise as soon as possible.

Because this exercise is based closely on the material on compilation presented in the Computer Science 1Bh lecture notes you should re-read those notes thoroughly to deepen your understanding of the concepts which you will need to understand in order to complete this practical exercise. The most profitable time to do this re-reading is before you begin any implementation work on the practical.

As with all Computer Science 1 practicals, this exercise cannot be undertaken in collaboration with any other student. However, there are several places where you can request help if you are having difficulties.

- From your Computer Science tutor. Your tutor will advise you on your general approach to the exercise and explain the concepts of the exercise at a high level.

- From the laboratory demonstrators. The laboratory demonstrators in the Level 3 laboratory of the Appleton Tower will help you with the specifics of your implementation. They will help with a lower level of detail such as explaining diagnostic error messages received from the Java compiler.

- From the CS1 Help Line. The CS1 Help Line is an email service whereby CS1 students can email specific questions about the practical exercise. These questions will be answered by CS1 tutors and demonstrators. This service is particularly relevant for people undertaking the practical exercise at home. To use this service send an email message to cs1help@dcs.ed.ac.uk explaining your problem thoroughly. You may add your implementation so far as an email attachment if this will help to clarify your explanation.

Stephen Gilmore.